An Integrated OS- plus VMM-bypass Solution for Virtualized I/O

Matthew Kurjanowicz (mkurjano@cc.gatech.edu)
Advisors: Ada Gavrilovska (ada@cc.gatech.edu)
Karsten Schwan (schwan@cc.gatech.edu)

May 2, 2007

Abstract

Current Remote Direct Memory Access (RDMA) technologies are either not virtualization aware or run on proprietary connection fabrics such as InfiniBand [8], Myrinet [23] and Quadrics [25]. The introduction of 10-Gigabit Ethernet provides a high-speed connection that utilizes existing network infrastructure. As the first step to provide a useful, virtualization-aware RDMA implementation we propose an integrated OS- and VMM-bypass solution implemented on the Netronome NFE-i8000 Network Processor utilizing the Intel IXP 2855. We use this implementation to demonstrate the feasibility and usefulness of an integrated VMM- and OS-bypass communications engines that can be used to implement such a virtualized RDMA solution and to understand the architectural limitations of such an implementation.

1 Introduction

Today's computer systems and applications are becoming increasingly network dependent. These applications must communicate vast amounts of information while remaining reliable and power-efficient. To this end, the network infrastructures over which these systems communicate are very important. These infrastructures can be divided into two groups: Local Area Networks (LANs) and Wide Area Networks (WANs). A local area network may contain all the computers in a server room, the floor of a building, or even a building. Interconnects used in a LAN environment typically provide extremely good performance, but are often limited by the distance over which data may travel. A wide area network may contain a set of buildings, or a set of computers spread over the world – the Internet is a good example. WANs face problems that LANs do not face, mainly in the areas of reliability and distance. Because of this, the infrastructures used in WAN environments cannot offer the same level of performance that can be offered in the LAN environment.

With CPU processing power growing exponentially, I/O subsystems are becoming the bottleneck [5] in communications performance. Proprietary interconnects, such as InfiniBand [8], Myrinet [23], and Quadrics[25] provide speedy interconnects, but can only be used in local area networks. These interconnects provide both high bandwidth and very low latency. Even though these high performance options exist,

Organizations are looking for ways to leverage their existing Ethernet infrastructure while meeting demands for higher performance. With the advent of 10-Gigabit Ethernet this is possible in both LAN and WAN environments [10, 4] without the significant capital outlay required by moving to an interconnect such as InfiniBand.

These communication bottlenecks become magnified in a virtualized environment. A virtualized environment is one in which more than one instance of an operating system run on one physical computer. Virtualization is making a resurgence in the data center by reducing cost [29] while retaining the security isolation guaranteed when not using virtualization [3].

In a virtualized environment, Raj et al. show that adding virtualization support to Ethernet devices increases performance dramatically [26] and Liu et al. show that RDMA in a virtualized environment is possible[18]. We provide an alternate implementation of an Operating System and Virtual Machine Monitor Bypass over Ethernet. Such a solution provides high-performance I/O in a virtualized environment.

2 Background and Related Work

In this project we attempt to improve the speed of the communication between computers over a network in a virtualized environment. There has been significant work in academics and industry up to this point. Here we discuss some of this work.

2.1 Operating System Bypass

One way of improving the performance of an application that accesses devices is to bypass the operating system kernel. In the general case the kernel must mediate all operations that access a device. For instance, if an application wishes to read from the memory on board a device, the application would request the information from the kernel. The kernel would then read the memory of the device and copy that into a buffer in the kernel. This is called a bounce buffer. The kernel then copies the data into the application's memory space.

While the bounce buffer implementation is inefficient, it is reliable and secure. The reason this is necessary is because many devices are unaware of the specific processes running in the operating system, nor can these devices write to the entirety of the host's RAM. The kernel provides the abstraction between the process and the device. However, if the device did know about the processes and the processes knew about

the devices the kernel could be avoided. This would avoid an expensive[15] memory copy.

In the networking domain, Remote Direct Memory Access (RDMA) enabled devices work by taking data directly from one applications' buffers, sending that data across a network and then placing that data directly into the application buffers on a remote host [27].

Many applications benefit by using RDMA devices. Some applications such as those that use MPI [19] and Grid technologies [24] lie in the high-performance arena. Other applications lie in the data center. Network storage [14] and web servers [28] are examples.

RDMA enabled network devices exist in academics and on the market. An early example is the Memory-Integrated Network Interface which locates the memory interface on the memory bus, thus giving it direct access to memory[22]. This network interface was able to transmit and receive data at 1.12Gbits/s over a 1.5GBaud optical connection, an impressive number in 1995. This approach, however, required significant changes the hardware layout of a computer, which has not become popular. On the market today, there are specialized interconnects that support RDMA, such as InfiniBand[8], Quadrics[25], and Myrinet[23]. These interconnects provide extremely high-performance networking at the cost of physical infrastructure. Each interconnect requires its own cabling, switching, and host adapters – often at significant monetary cost and making the deployment of these interconnects in a Wide-Area Network difficult or impossible. This cost can be avoided by supporting RDMA implementations on an Ethernet NIC, such as the Ammasso Gigabit Ethernet NIC[14]. While not having the raw performance of the specialized interconnects we mentioned previously, an RDMA enabled Ethernet NIC has the distinct advantage of utilizing the existing infrastructure of data centers today.

2.2 Protocol Offload Engines

Traditional implementations of network protocol stacks consume significant CPU cycles [15]. These implementations often reside in the operating system kernel and require data to be copied from application to kernel buffers.

Placing this processing on the network interface card (NIC) removes much of the burden from the CPU while still allowing applications to use the standard `socket` API. This avoids overheads from system calls, hardware interrupts and context switches and therefore lowers the latency and overhead of communications [6].

Even though this offload reduces CPU usage, there are downsides. First, per-connection resources must

be managed on the NIC. Hardware that could support many concurrent connections would be prohibitively costly. Some solutions implement a negotiation between the hardware and software protocol stacks that places some of the burden back into the kernel when too many connections are established [16].

Other implementations of protocol offload place the network interface on the memory bus instead of the I/O bus. The Memory-Integrated Network Interface [22] is an example that implements the ATM protocol. Other protocols such as TCP can be implemented on top of this.

2.3 Programmable Communication Engines

Some NICs are programmable. Such hardware is called programmable network processors (NPs). Since NPs cannot use the cache locality so heavily depended upon by general purpose processors, they are not as efficient at accessing application buffers as the CPU [31]. Instead these processors benefit by residing close to the physical network connection.

NPs add additional services to networks, such as profiling and quality of service guarantees. They can be reprogrammed to support new protocols without investment in new hardware. NPs are also valuable development platforms.

One such series of NPs is the Intel IXP series[12, 13]. These processors consist of a general purpose Xscale processor in addition to specialized *microengines* (μEs). The μEs provide fast-path processing while the Xscale is used in slow-path processing. Each microengine contains eight hardware-level threads with minimal overhead context-switching. Each thread is known as a separate context. Context-switching is voluntary by each context.

2.4 Virtualization

In the loosest sense, Robert Goldberg, in his *Survey of Virtual Machine Research*[7], defines virtualization as running "many copies of a machine on itself." A virtual platform provides an interface from a physical machine or operating system to applications or other guest operating systems. An example of virtualization that provides an interface from an operating system to an application is the Java Virtual Machine (JVM). Applications written in Java are compiled to a bytecode that the JVM can understand. The JVM, in turn, runs as an application on the operating system. This allows applications written in Java to have a (mostly) consistent interface over many operating systems and architectures. In this paper, we are more interested

the other part of this definition – virtual machines that provide an interface between a physical computer or operating system to a guest operating system.

Initially developed in the 1960s to provide concurrent multiuser access to a shared mainframe computer, virtual machines evolved to being used to address problems in data security, reliability, and software development by the early 1970s [3, 7].

Today virtualization is deployed in the data center. It allows for security isolation and the ability to guarantee certain qualities of service on a per-machine basis. To achieve the same guarantees when running each application on multiple machines would cost significantly more in terms of space, power, cooling and maintainability [29].

Modern operating systems assume they have full, direct access to the machines they run on. This assumption is no longer valid when running in a virtual environment. Instead there must be some software responsible for sharing the computer's resources among the virtual machines. This software is called the Virtual Machine Monitor (VMM).

One area in which virtualization causes significant performance degradation is I/O. Consider a virtualized system with two guest domains and both domains heavily access a single physical Ethernet connection. The VMM must then multiplex all incoming and outgoing data. In a situation like this, I/O processing quickly becomes CPU bound. Sugarman [30] shows that I/O processing becomes CPU-bound in high-traffic devices such as network interfaces due to the multiplexing required by the VMM.

In this article, we consider Xen [36] – one specific implementation of virtualization. Xen controls the hardware through a special virtual machine known as a device domain. Communication from a virtual machine to Xen is through synchronous hypercalls. Communication in the reverse direction is via a lightweight event-based notification mechanism similar to hardware interrupts [2]. Besides Xen, other virtualization solutions include KVM[17], Microsoft VirtualPC[21] and Virtual Server[20], VMware ESX Server[33], Workstation[35] and Server[34], and Virtuozzo[32].

Xen provides a mechanism called page-switching. By this mechanism, two virtual domains, say `dom0` and a `domU`, can share data by mapping memory pages from `dom0` into the memory space of `domU`. Traditionally, this would be done by copying the data in memory between the two locations. Swapping pages obviates these memory copies.

2.5 Virtualized Network I/O

Current work in the field is directed at improving the performance of I/O in a virtualized environment. Without hardware support, all I/O data must traverse from a device to the specific virtual machine's kernel through the VMM. Switching to the VMM incurs significant performance overheads. Supporting virtualization on the device avoids most of this overhead.

Current network hardware that supports virtualization is rare. Therefore, current and recent research has focused on implementing self-virtualization using programmable network engines, such as the IXP2400[12], or by utilizing intrinsic capabilities of hardware such as InfiniBand[8].

2.5.1 SV-NIC

A recent technical report[26] by Raj et al. presents the concept of Self-Virtualized I/O (S-VIO). This report describes an abstraction of an I/O device that, in their words, virtualizes itself. We can see in Figure 1 that this I/O devices consists of four logical components: the processing component consisting of any number of computation cores; the physical device which is the actual device presenting itself; the Peripheral Communication Fabric, which connects the processing component to the physical devices and could be either a dedicated fabric or a shared fabric such as PCI; and finally the Messaging Fabric, which connects guest domains to the virtual device. This framework could be used to virtualize any device by utilizing one or more, possibly heterogeneous, cores as the processing component.

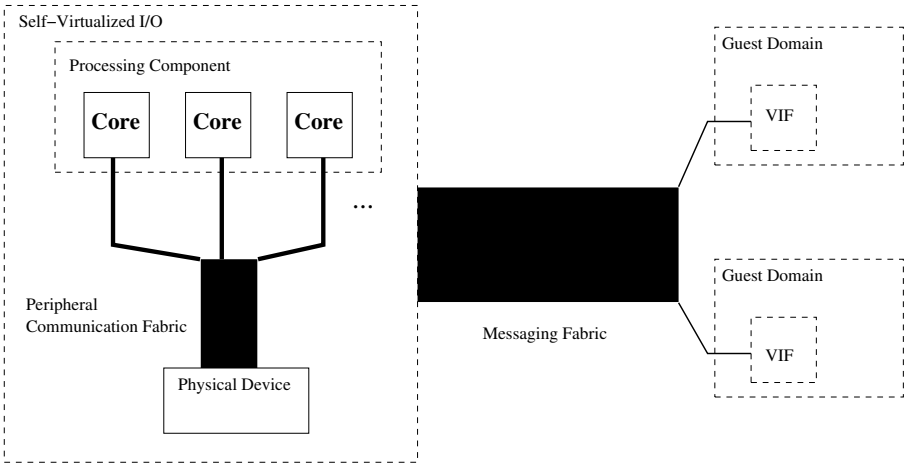

Figure 1: The S-VIO Abstraction [26]

The group presenting the S-VIO abstraction realized this in a high-performance networking device – the

Self-Virtualized NIC (SV-NIC) illustrated in Figure 2. They implemented the SV-NIC on the IXP 2400[12] programmable network processor. In this implementation, the host and the IXP create virtual tunnels between guest domains and the IXP. These tunnels are constructed by `dom0` which then passes this information of to a guest domain. All further communication between the guest domain and the IXP then proceeds with minimal use of `dom0`. These tunnels are named Virtual Interfaces (VIF)s.

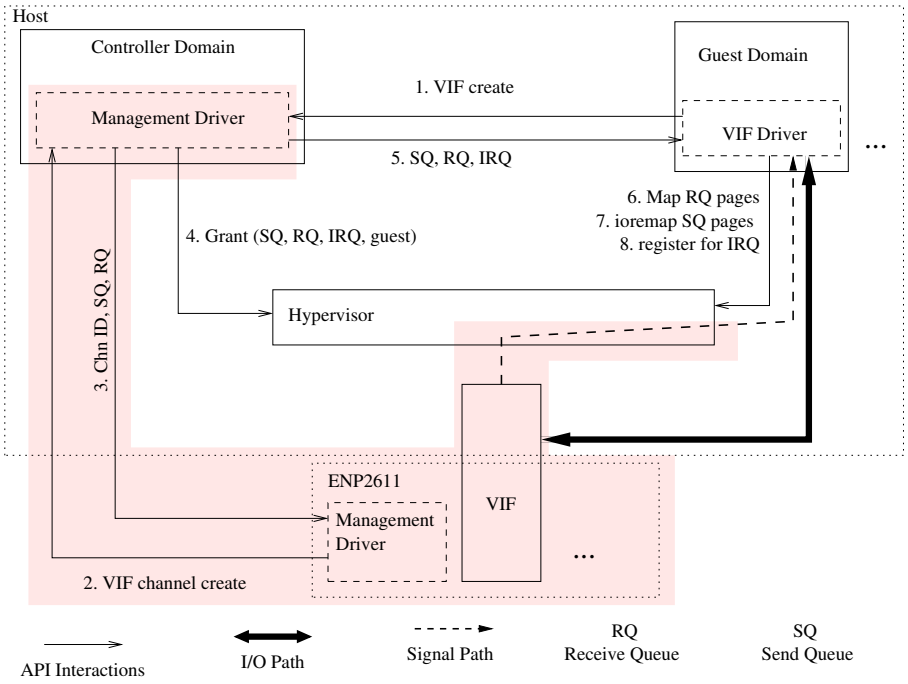

Figure 2: The SV-NIC. [26]

Each VIF transfers packets between the guest domain and the IXP in two directions. The directions are named ingress and egress from the IXP point of view. Ingress manages packets from the network wire and demultiplexes them into the correct VIF tunnel. Egress manages packets from the guest domain to be sent out over the wire. For balance purposes, every VIF is allocated one micro-engine context for egress. A shared pool of microengines is managed for ingress.

The SV-NIC implementation is limited by the bus connection between the host and the IXP. The IXP2400 is located on the PCI bus using an Intel 21555 Non-transparent PCI-to-PCI bridge[9]. This limits the amount of host memory the microengines can address to sixty-four one-Megabyte regions[26]. Because of this limitation, it is impossible to implement OS-bypass in addition to VMM-bypass on the IXP2400.

Raj et al. show significant performance improvements with their SV-NIC implementation. When com-

pared to host-based virtualization, the bandwidth of the SV-NIC is 200 percent faster, even with high numbers of guest domains. The SV-NIC does show an improved latency with up to sixteen guest domains, but slower latency with more than sixteen domains. This is due to their implementation of IRQ (Interrupt Request) sharing.

I/O devices can communicate with the host in two ways – memory polling and IRQs. With memory polling, the device will set some value in host memory or the host will set some value in device memory. The device or host will then check this memory location at some time interval, and note if the value changes.

IRQs provide a way for an I/O device to interrupt the host or for the host to interrupt the device. When an interrupt is sent, the device or host may stop what it is doing and handle this interrupt. On the IXP series, microengines are not preemptable and therefore will not receive interrupts.

PCI devices have access to only one sixteen-bit wide interrupt. An interrupt is sent to the host when any one of these sixteen bits is changed. The SV-NIC implementation further divides this into eight regions, so that a reason for the interrupt may be communicated to the host. Thus, when the SV-NIC wishes to notify the host that some I/O operation is complete via an interrupt, it may only choose between any of these eight fields, so virtual tunnnels must share these fields. These IRQs must then be interpreted by the VMM. Xen provides the ability to virtualize these IRQs.

2.5.2 VMM-Bypass on InfiniBand

Another implementation of VMM-bypass utilizes InfiniBand. The implementation by Liu et al.[18] gives similar benefits as the SV-NIC implementation discussed previously. However, this implementation uses the InfiniBand interconnect.

There are limited, but significant drawbacks to the VMM-bypass over InifiniBand implementation. This implementation does not lie in the Ethernet domain. Organizations are currently looking for ways to leverage their existing Ethernet infrastructure. The advent of Ten-Gigabit Ethernet has made this a possibility [10, 4].

The Liu et al. VMM-bypass implementation is able to leverage the immense bandwidth and low latency of the InfiniBand interconnect. As a result, they have been successful in implementing traditional high-performance-computing libraries, such as MPI, to their VMM-bypass. Furthermore, they are able to implement RDMA over InfiniBand. This provides both a VMM- and OS-bypass.

Because of the ability to leverage the performance of InifiniBand, Liu at al. created an implementation which could eventually be used in the high-performance industry. This industry generally cares only

about performance, and there is certainly a loss of performance in most virtualized systems. However, this performance loss could be offset in the future by different programming paradigms.

Liu et al. do not evaluate the performance of their implementation on multiple domains, an important performance metric for any virtualization-based solution. Their performance evaluation focuses on the performance of a single guest domain versus the performance of multiple domains.

3 Implementation

In the previous section, we discussed two implementations of self-virtualizing devices – the InfiniBand solution by Liu et al.[18] and the self-virtualized NIC by Raj et al.[26]. In this section, we will present another implementation, a self-virtualized Netronome NFE-i8000. We utilize the expanded memory-access capability of the NFE-i8000 to provide a framework for communications between the host and the IXP that bypasses the domains' kernels and the Xen hypervisor. Our first attempt to implement this solution used the already existing Netronome Flow Driver provided by Netronome Systems with their NFE-i8000. However, this driver required a host-side kernel driver to multiplex and demultiplex messages between the host and IXP. This would require a world swap for every message sent, which we avoid with our implementation.

The communications engine contains multiple drivers on the host and IXP, as well as clients on the host and IXP. These clients are the endpoints for the messaging system, shown in Figure 3. A host-side client communicates to an IXP-side client over a virtual channel. Each channel contains two queue pairs and a set of control registers on the device, shown in Figure 4. One queue pair resides on the host and the other on the NFE. These pairs contain one queue in each direction. The host pair is set up so that the send queue contains messages from the host to the IXP and the receive queue is the opposite. Likewise, on the IXP the send queue contains messages from the IXP to the host and the receive queue messages from the host to the IXP.

The host side implementation consists of a few drivers. The first driver, named gtixp, is loaded into a privileged kernel (either in `dom0` or a non-virtualized kernel) and maps device I/O regions into CPU memory space. One such region is BAR0, which is a set of eight byte aligned, four byte control registers on the NFE. These control registers allow drivers to send small bits of information between the host and IXP. Once mapped into CPU memory, these registers appear like normal system memory to the kernel, but operate in a different manner. Every time a register is read or written, a notification is sent to the NFE. A

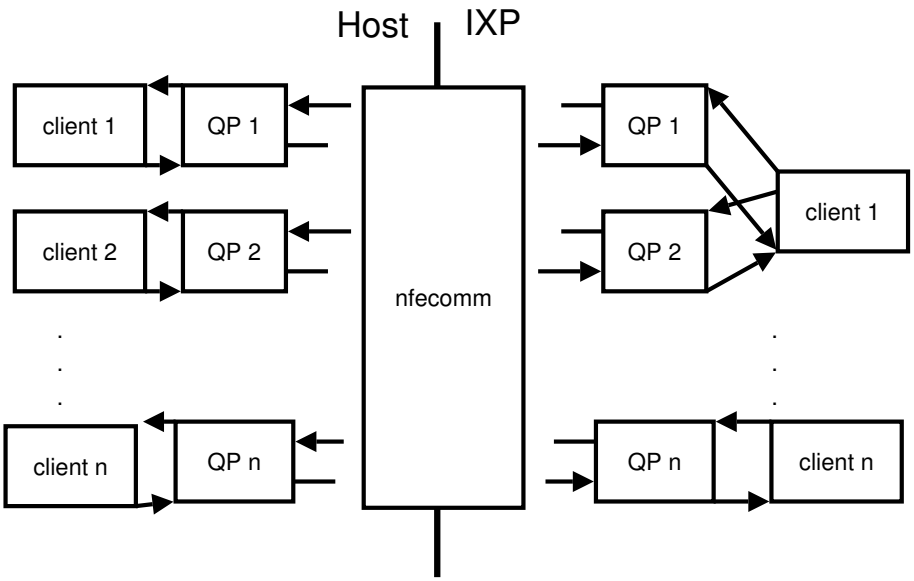

Figure 3: A conceptual overview of the communications channel. Each client communicates with respective clients on the other device. Note that a client may subscribe and write to multiple message queues. These queue pairs are mirrored in the opposite device's memory.

microblock on the NFE must then poll a queue for these notifications, or tags, and respond appropriately. If the host read that register, this microblock must write the correct value. If the host wrote to that register, the microblock must read the value and store it appropriately. Thus, all values contained in these registers are stored on the NFE and can change at any time without the host knowing.

The nfecomm driver contains components that allow the host to send data to the IXP and for the IXP to send data to the host. These components are loaded depending on the context. On a non-virtualized system, the non-virtualized control and communications components are loaded. On the privileged domain of a virtualized system, these components are loaded as well as a component that provides control-path communication over XenBus with other domains. In a virtualized environment on non-privileged domains, the communications and front-end control components are loaded.

The nfecomm control component bootstraps a control channel and creates and destroys new channels. To bootstrap the driver, the control component first allocates memory for a channel in host memory. It then polls a control register to see if the FPGA has been programmed by the nfecomm component on the IXP. Once the FPGA has been programmed, the nfecomm control driver notifies the nfecomm IXP component, through control registers, the location of the first channel in host memory. This channel is then used for all further control path communication.

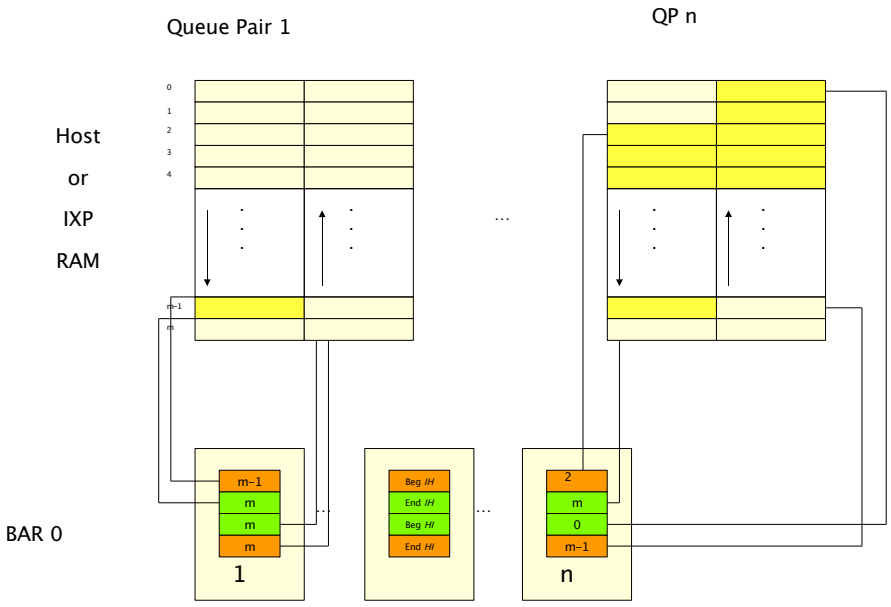

Figure 4: An example of the layout of a queue pair. Each queue pair is located inside its segment of memory. The control pointers are conceptually located on BAR0, but physically reside on the IXP.

To create a new channel at runtime, the nfecomm control component first allocates memory in the host and sends a control message to the IXP component. This, like all other message sends in the system, is asynchronous. The IXP nfecomm component then allocates memory in the IXP SDRAM for the messages, then sends a signal to a special client of the IXP-side messaging system telling this client about the new channel. The client then performs any client-specific initialization of its state and notifies the nfecomm control microblock. The nfecomm control microblock then sends a message to the host via the control channel. While these control requests are being processed by the IXP, the host-side component maintains a queue of pending requests, which can be polled by the client at any time. These requests are in any one of three states: Pending, Success, and Failure. Pending indicates that the request has been sent to the IXP, but the driver has not received a message back. Success implies that the channel is successfully created and the client can use the communications component to send and receive data between the IXP. Failure implies that the IXP could not set up the channel, therefore no host resources are allocated and the client should not attempt to send data over this channel.

The nfecomm communications component contains all host-side utilities for sending messages to and receiving messages from the NFE. It allows a client to allocate and send a message to the NFE and poll

for messages from the NFE. There is a similar component on the IXP. There are two kinds of messages that the nfecomm driver supports. Both types of message are sixty-four bytes and contain some header information and a data section. The first type of message, a user-data message, contains these headers and some client-defined data in the data section of the message. When the nfecomm microblock receives this message, it programs the FPGA the message directly from host memory to the appropriate location in the client's receive queue. The other type of message, a DMA-data message, contains the address and size of some data in the host. When the nfecomm microblock receives this message, it programs the FPGA to DMA this data from the host to the IXP. When this DMA of the data is complete, the nfecomm microblock marks a header in the data message as complete. Only then can the client reliably consume the message.

To send data from the IXP to host, the IXP client can choose between the same two types of messages as the host. Sending a user-data message is simple, as the nfecomm microblock programs the FPGA to DMA the message to host memory. Transferring large amounts of data is not as simple. The IXP client must maintain locations of data that it can write to for each channel. Currently, the only locations that the client DMAs data to are located within the channel memory allocated by the host. This could pose a serious security risk, as any client can DMA data to any location in host memory, possibly overwriting other data, including operating system routines.

The nfecomm driver also contains an IXP component. This component consists of two microblocks. Each microblock runs on one of the sixteen microengines on the NFE's Intel IXP2855 network processor. These components are the DMA Read and DMA Write microblocks. The DMA Read microblock is responsible for handling all messages sent from the Host to the IXP. The DMA Write microblock handles communication in the opposite direction. The DMA read microblock also handles the control path. These microengines store various pieces of data in different locations of the IXP memory hierarchy. Local memory, which is specific to each IXP, is used to store temporary DMA state information. Scratch memory, which is a small 16 Kilobyte region shared among all microengines, is used to store the queue pointers that the host accesses through BAR0. Finally the microblocks use SRAM, which is a much larger portion of memory that is much slower, to store the messages that are to be transferred to the host. Data that the messages point to, such as packets, are stored in DRAM.

The DMA Read and Write microblocks communicate with the FPGA via a memory region mapped into the SRAM memory space. All FPGA operations are asynchronous, so each microblock must poll for, and interpret, messages called tags. These tags contain status and control information, such as when a DMA

operation is completed. Because these operations are asynchronous, the DMA Read and Write microblocks each have a queue for storing pending DMA operations. Each of these microblocks is responsible for handling a certain aspect of the control registers logically stored in BAR0. The DMA Read microblock handles host-writes of these registers and the DMA Write microblock handles host-reads of these registers. If the DMA Read or DMA Write microblock polls for, and processes, a tag indicating an operation by the host that the microblock should not handle, it communicates this request over the IXP built-in next-neighbor communication ring.

An example application on the IXP is shown in Figure 5. This diagram gives a conceptual picture of the NFE. The Tx (transmit) and Rx (receive) microblocks transfer packets to and from the the IXP Tx/Rx buffers via the IXP Media Switch Fabric. One microblock will then poll the Rx microblock for packets, this is the IXP to Host microblock. Based on the MAC address of incoming packet, the IXP to Host microblock will enqueue a DMA-data message onto the channel for the domain that owns the particular MAC address. The DMA write microblock polls for messages on this queue and programs the FPGA to DMA the data from SDRAM into the Host RAM. In the opposite direction, the DMA Read microblock waits for messages from the host and programs the FPGA to transfer the data into IXP SDRAM. The Host to IXP microblock then polls each of the receive queues of each channel and notifies the Tx microblock to transfer the packet to the wire.

The nfecomm host-side control-path must be modified to run under a virtualized environment. The virtualized control path provides a back-end and front-end driver to create new channels. When a non-privileged guest VM wishes to create a new communications channel, the control front-end driver sends a request over XenBus to `dom0` to allocate the resources for a new channel. On `dom0`, the back-end driver listens over XenBus for these requests and uses the nfecomm control component to send a new driver channel request to the IXP. The back-end driver then waits for the IXP to process the request and send a Success or Failure notification to the host. If the creation is successful, the back-end driver will then grant access to the channel memory and a page in BAR0 to the requesting guest VM. Finally, the back-end driver sends a message to the requesting VM via XenBus informing this VM of the location of these pages of memory. If the channel creation fails, the back-end driver notifies the guest VM, via XenBus, of the failed attempt. A guest VM can destroy its own channels in a similar manner. Because the communications component only modifies data within a single channel, once a single channel is allocated to a guest VM the communications component behaves no differently in virtualized verse a non-virtualized environment. Since each guest VM

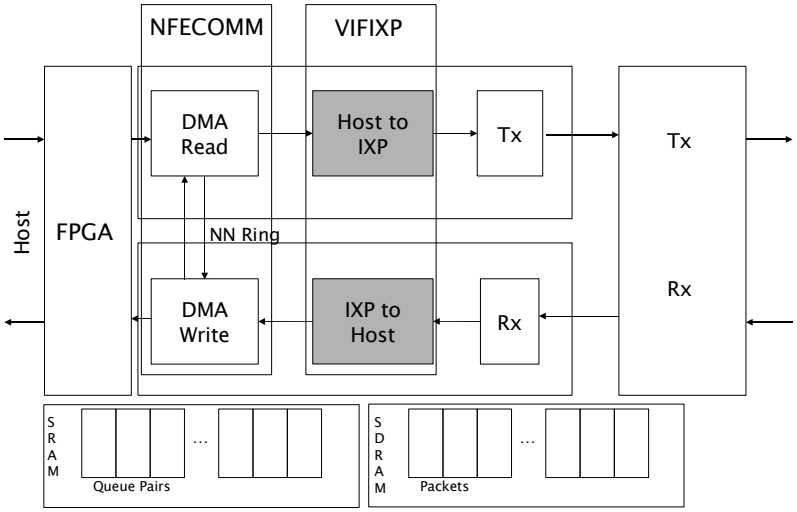

Figure 5: An example application of the nfecomm driver on the IXP, a Virtualized NIC. In this case, each microblock runs on one microengine. This example shows one client on the IXP for many clients on the host.

has at least its own channel, the IXP components do not need to change in a virtualized environment, and besides for control path communication, no world switch is required to send data to or receive data from the IXP.

Our hardware setup consists of two identical machines. Each machine contains one dual-core Pentium D CPU at 3.20GHz with 3GB of RAM. Each contains one Netronome NFE-i8000, which contains one Intel IXP 2855 Network Processor. Each NFE has four Gigabit Ethernet ports. The first port on one processor is connected via a patch cable to the first port on the other. Each has Fedora Core 6 with all software updates installed. We test our driver in both Linux 2.6.20 without virtualization and Xen-current version 11269. We were unable to gather any performance results.

4 Conclusion

While implementing this communications driver, we made many design decisions that often compromised speed and performance for simplicity. The most major of these involves the way that messages are transferred from the host to the IXP. Since the host only communicates the indices of the messages in the queues

to the IXP, the IXP is responsible for queuing the actual transfers of the messages from the host. Due to the general nature of our implementation, the IXP does not know how many messages it will receive, so it enqueues a new DMA descriptor for every message transfer the host indicates. If the host notifies the IXP after queuing every single message, the IXP would be constantly queuing DMA descriptors to transfer sixty-four bytes. This is a lot of overhead for each message, which shows when the communications engine is stressed under moderate and high loads. To avoid this, the host should only notify the IXP of new messages after every few messages are enqueued by the host side, this would mitigate some of the overhead. In addition, we chose to allocate two megabytes of memory for each channel, 256 slots in each queue, and a message size of sixty-four bytes. These sizes should be empirically tested to find the optimal message size and number of messages per queue to reduce congestion.

Another decision we made was to only have the host and IXP poll for messages received. This polling takes up many resources, so the user of the driver must strike a balance between polling frequency and latency. The higher the polling frequency, the lower latency will become, but the more system resources will be avoided. If the polling frequency is too low, then the queues may become backed up and packets could be dropped.

5 Future Work

We have presented a solution that future researchers can use as a core component in other research projects. This communications engine must first be fully debugged. Then this engine could be used to implement a self-virtualized device akin to Raj's SV-NIC. From that point, these projects could span from traditional high-performance computing applications, such as MPI implementations, to data center applications, such as NFS over RDMA.

References

[1] Amd amd-v.

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Oystems Principles*, pages 164–177, Bolton Landing, New York, USA, 2003. ACM Press.

[3] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, pages 133–138, Schloss Elmau, Germany, May 2001. IEEE Computer Society.

[4] Wu chung Feng, Justin (Gus) Hurwitz, Harvey Newman, Sylvain Ravot, R. Les Cottrell, Oliver Martin, Fabrizio Coccetti, Cheng Jin, Xiaoliang (David) Wei, and Steven Low. Optimizing 10-Gigabit Ethernet for networks of workstations, clusters, and grids: A case study. In *Proceedings of the Supercomputing Conference*, Phoenix, Arizona, USA, November 2003.

[5] David D. Clark and David L. Tennenhouse. Architectural considerations for a new generation of protocols. *ACM SIGCOMM Computer Communication Review*, 20(4):200–208, September 1990.

[6] Dennis Dalessandro, Ananth Devulapalli, and Pete Wyckoff. Design and implementation of the iWARP protocol in software. In S. Q. Zheng, editor, *Proceedings of the 17th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS '05)*, pages 471–476, Phoenix, Arizona, USA, November 2005. ACTA Press.

[7] Robert P. Goldberg. Survey of virtual machine research. *IEEE Computer*, 7(4):34–45, June 1974.

[8] InfiniBand Trade Association. Online: `http://www.infinibandta.org/home`.

[9] Intel 21555 non-transparent PCI-to-PCI bridge. Online: `http://www.intel.com/design/bridge/21555.htm`.

[10] 10-gigabit ethernet expands network bandwidth and shrinks latency. Technical report, December 2005.

[11] Intel vt-x.

[12] Intel IXP2400 network processor. Online: `http://www.intel.com/design/network/products/npfamily/ixp2400.htm`.

[13] Intel IXP2855 network processor. Online: `http://www.intel.com/design/network/products/npfamily/ixp2855.htm`.

[14] Hyun-Wook Jin, Sundeep Narravula, Gregory Brown, Karthikeyan Vaidyanathan, Pavan Balaji, and Dhabaleswar K. Panda. Performance evaluation of RDMA over IP: A case study with the Ammasso Gigabit Ethernet NIC. Technical Report OSU-CISRC-6/05-TR40, Ohio State University, Columbus, Ohio, USA, 2005.

[15] Jonathan Kay and Joseph Pasquale. The importance of non-data touching processing overheads in TCP/IP. In *Proceedings of the Conference of the Special Interest Group on Data Communication (SIGCOMM '93)*, pages 259–268, Ithica, New York, USA, September 1993. ACM Press.

[16] Hyong-Youb Kim and Scott Rixner. TCP offload through connection handoff. In *Proceedings of the 2006 EuroSys Conference*, Leuven, Belgium, April 2006.

[17] Kvm: Kernel-based virtual machine for linux. Hosted, Full virtualization (with VT-X or AMD-V). Online: `http://kvm.qumranet.com/kvmwiki`.

[18] Jiuxing Liu, Wei Huang, Bulent Abali, and Dhabaleswar K. Panda. High performance VMM-bypass I/O in virtual machines. In *Proceedings of the 2006 USENIX Annual Technical Conference*, pages 29–42, Boston, Massachussetts, USA, May 2006. The USENIX Association.

[19] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K. Panda. High performance RDMA-based MPI implementation over InfiniBand. *International Journal of Parallel Programming*, 32(3):167–198, June 2004.

[20] Microsoft virtual server. Hosted, Full virtualization. Online: `http://www.microsoft.com/windowsserversystem/virtualserver/default.mspx`.

[21] Microsoft VirtualPC. Hosted, Full virtualization. Online: `http://www.microsoft.com/windows/products/winfamily/virtualpc/default.mspx`.

[22] Ron Minnich, Dan Burns, and Frank Hady. The memory-integrated network interface. *IEEE Micro*, 15(1):11–19, February 1995.

[23] Myricom, inc. Online: `http://www.myri.com/`.

[24] Hidemoto Nakada, Satoshi Matsuoka, Keith Seymour, Jack Dongarra, Craig A. Lee, and Henri Casanova. A GridRPC model and API for end-user applications. Technical Report GFD-R.052, GridRPC Working Group, July 2005.

[25] Quadrics, ltd. Online: `http://www.quadrics.com/`.

[26] Himanshu Raj, Ivan Ganev, Karsten Schwan, and Jimi Xenidis. Self-Virtualized I/O: High performance, scalable I/O virtualization in multi-core systems. Technical Report GIT-CERCS-06-02, Georgia Institute of Technology, Atlanta, Georgia, USA, 2006.

[27] Allyn Romanow and Stephen Bailey. An overview of RDMA over IP. In *Proceedings of the First International Workshop on Protocols for Fast Long-Distance Networks (PDFLnet '03)*, Geneva, Switzerland, February 2003.

[28] Allyn Romanow, Jeffrey C. Mogul, Tom Talpey, and Stephen Bailey. RFC 4297: Remote direct memory access (RDMA) over IP problem statement. Technical report, IETF Network Working Group, December 2005.

[29] Mendel Rosenblum and Tal Garfinkel. Virtual machine monitors: Current technology and future trends. *IEEE Computer*, 38(5), May 2005.

[30] Jeremy Sugarman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, Massachussetts, USA, June 2001. The USENIX Association.

[31] Muthu Venkatachalam, Prashant R. Chandra, and Raj Yavatkar. A highly flexible, distributed multiprocessor architecture for network processing. *Computer Networks*, 41(5):563–586, April 2003.

[32] Virtuozzo. Hosted, Full Virtualization. Online: `http://www.swsoft.com/en/virtuozzo/`.

[33] Vmware ESX server. Non-Hosted, Full virtualization. Online: `http://www.vmware.com/products/vi/esx/`.

[34] Vmware server. Hosted, Full virtualization. Online: `http://www.vmware.com/products/server/overview.html`.

[35] Vmware workstation. Hosted, Full virtualization. Online: `http://www.vmware.com/products/ws/overview.html`.

[36] The Xen virtual machine monitor. Non-Hosted, Paravirtualization or Full virtualization with VT-X[11] or AMD-V[1]Online: `http://www.cl.cam.ac.uk/research/srg/netos/xen/`.